BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

The Spring Name Service

Sanjay Radia
Michael N. Nelson
Michael L. Powell

SMLI TR-93-16 November 1993

Abstract:

The Spring name service exploits and supports the uniformity of objects in the Spring object-oriented distributed system. The name service can be used to associate any name with any object independent of the type of object, and arbitrary name spaces can be created and used as first-class objects. The same client interface is used not only for all conventional operating system and network naming, but also for other services that support naming-style interaction. The architecture of the name service is open, supporting combinations of trusted and untrusted name servers and object implementations.

The name service integrates access control and persistence for objects in a way that allows object implementations to delegate responsibility to the name service, or to implement their own policies. The interfaces between different name servers and between name servers and object implementations allow a variety of implementation strategies for objects and name servers, providing different levels performance, persistence, robustness, security, and convenience.

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
sanjay.radia@eng.sun.com
michael.nelson@eng.sun.com
michael.powell@eng.sun.com

The Spring Name Service

Sanjay Radia, Michael N. Nelson, and Michael L. Powell

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

An operating system has various kinds of objects, such as files, printers, services, etc., that need to be given names. Most operating systems have several name services, each tailored for a specific kind of object. Such *type specific* name services are usually built into the subsystem implementing those objects. For example, a file system typically includes its own mechanism for binding names to files, accessing files by name, etc., and manages the database of those bindings. In the UNIX ®[1] operating system, the name space for printers is stored in */etc/printcap*, and is used by the various printing commands. Lightweight name services such as environment variables in UNIX[8] have their own name space with library implementations. Type specific name services have interfaces that are designed for objects of the specific type, and are unsuitable for general naming. Distributed systems usually have one or more higher level name services called *directory* services, such as Grapevine[3], Lampson's global name service[2], or X.500[6], which are used for resource location, mail addressing, and authentication. Directory services bind names to data values that have to be resolved at another level.

In systems with such diverse name services, the client has to deal with different names and name services, depending on the objects being accessed. Often, separate parts of the name space must be used for different types of objects since, for example, it is not possible to bind a file as an environment variable. Another problem is that the non–uniform name spaces cannot generally be composed together. For example, the name spaces for files cannot be attached to the name spaces for environment variables. Defining new objects or services that are accessed by name can be complicated and cumbersome, since the new objects must be made to "fit in" somewhere, or a new name space must be defined and implemented.

Difficulties in Providing a Uniform Name Service

Providing a uniform name service has been difficult because the entities being named have different representations. Type specific name services provide tokens that must be used for a particular purpose, since the type is assumed; for example, in UNIX, looking up a file returns an integer, which is used for file operations. On the other hand, directory services provide values that must be resolved at another level to be useful; for example, looking up a host returns a byte string that can be interpreted as a network address.

One solution is to pick a particular object type, map all objects to provide that type's interface, and make all the object implementations (called object managers in Spring[2]) implement the type specific name service. This approach is popular in UNIX, where various objects are made to look like files, since the file and the file name service are the most versatile parts of the system. There are several problems. Unfortunately, it is difficult to make every object behave like a file, since the file interface is limited in capturing the semantics of many objects. For example, Plan 9[5, 4], which uses this approach extensively, still has a separate name service to find file servers. Another problem with this approach is that each object manager

1. UNIX is a registered trademark of UNIX System Laboratories.
2. Spring is an internal code name only.

must still implement its own (file) name service, which makes it difficult to introduce new object types into the system. A more detailed comparison with Plan 9 is provided in Section 8.

The uniformity of objects in object oriented systems offers an opportunity for a uniform name service. However, existing object oriented systems either have not provided a uniform name service or have provided one with serious restrictions. Programming language based object oriented systems such as Smalltalk rely on the name space of variables provided by the programming language. Systems like Choices[10,10] have the serious restriction that the object managers—the servers that implement the objects—must be integrated and reside with the name service. This makes it difficult to add new object types.

One of the main difficulties in building a uniform name service has been that the service often needs to interact with the object managers for the purpose of persistence and security. Type–specific name services such as the file system can address these problems easily, since the implementations of the name service and the objects reside together. Thus, for example, the name service implementation knows how to turn live files into those stored on persistent storage. Furthermore, there is complete trust between the two components to provide the user with a uniform view of access control—the name service can check the access control list on a file just like it checks the access control list on a directory. Directory services avoid the problem by simply binding names to data values, leaving the client to interpret the values. If, for example, a data value represents a way to get to a service, then it is up to the client to ensure that a persistent form (such as an address instead of a connection identifier) is bound, and that the value returned from a name resolution is turned into a live form when needed. Similarly, if the data value represents a way to get to a secure service, then the client has to perform additional authentication steps.

The Spring Name Service

Spring provides a uniform name service: in principle, any object can be bound to any name. This applies whether the object is local to a process, local to a machine, or resident elsewhere on the network; whether it is transient or persistent; whether it is a standard system object, a process environment object, or a user specific object. Name services and name spaces do not need to be segregated by object type. Different name spaces can be composed to create new name spaces.

By using a common name service, we do not burden clients with the requirement to use different names or different name services depending on what objects are being accessed. Similarly, we do not burden all object implementations with constructing name spaces—the name service provides critical support to integrate new kinds of objects and new implementations of existing objects into Spring. Object implementations maintain control over the representation and storage of their objects, who is allowed access to them, and other crucial details. Although Spring has a common name service and naming interface, the architecture allows different name servers with different implementation properties to be used as part of the name service.

Spring differs from existing name services in that its contexts and name spaces are first class objects: they can be accessed and manipulated directly. For example, two applications can exchange and share a private name space. Traditionally, such applications would have had to build their own naming facility, or incorporate the private name space into a larger system–wide name space, and access it indirectly via the root or working context.

The naming interface can be used even in subsystems whose primary purpose is not naming. For example, a spreadsheet application may support the naming interface to allow access to its cells by name. Like other name spaces, the spreadsheet name space could be attached to another name space. Transient name services, such as process environment variables, can also implement the same interface. Although different

implementations may be used to meet the different performance and distribution needs of these various name spaces, the interface is the same.

Spring object references are generally not persistent, and instead, naming is used to support the persistence of objects (we explain our reasons and approach in more detail later in the paper). It is expected that applications generally (re)acquire objects from the name service, which deals with the persistence aspect of objects if the name space happens to be persistent.[3] A Spring name server managing a persistent part of a name space converts the object references and objects to and from their persistent form (much like the UNIX file system, which converts open files to and from their persistent form). One difficulty in integrating persistence is that naming is a generic service for an open–ended collection of object types, and hence cannot be expected to know how to make each object type persistent. Spring objects are abstractions, so the object manager has ultimate control of the (hidden) state of the object. We provide a general interface between object managers and the name service that allows persistence to be integrated into the name service while allowing the implementation to control its storage.

Because the name service is the most common mechanism for acquiring objects, it is a natural place for access control and authentication. Since the name service must provide these functions to protect the name space, it is reasonable to use the same mechanism to protect named objects. The naming architecture allows object managers to determine how much to trust a particular name server, and an object manager is permitted to forego the convenience and implement its own access control and authentication if it wishes. Similarly, name servers can choose to trust or not to trust other name servers.

The distinguishing features of the Spring name service are a uniform and extensible name service, a general naming model with first class name spaces, and integrated support for access control and persistence. While different name services have provided some of these features in the past, the challenge in Spring has been to integrate all into a single name service.

This Paper

This paper describes the architecture of the Spring name service, focusing on its distinguishing features mentioned above. After the overview of the Spring system in Section 2, Section 3 presents the naming model on which the name service is built. Sections 4 and 5 present our general approach to persistence and security, and describe the framework for them, and this leads to a discussion in Section 6 of the interface and protocols used for interactions among clients, name servers, and object managers. Various examples from our system, including our naming policies, are described in Section 7. In Section 8, we make comparisons with other related work. Finally, Section 9 provides some concluding remarks.

2 An Overview of the Spring System

Spring is a distributed, object oriented operating system. A Spring object is an abstraction that is defined by an interface, expressed in the IDL interface definition language [18], which specifies the set of operations that may be performed on the object. IDL supports multiple inheritance of interfaces, and Spring encourages this, allowing objects to export their functionality via multiple interfaces. An interface is a strongly typed contract between the *implementation* (the *object manager*) and the *client* of the object. The granularity of Spring objects spans a wide range, from small data–like objects to larger objects, such as files and print services. The implementations vary from data accessors and libraries, to separate protected servers, to distributed services implemented by multiple servers.

---

3. In fact, we would implement persistent object references by registering objects in the name service, using a name server designed for that purpose.

Clients manipulate objects by performing operations on them, or passing them as parameters in operations on other objects. Generally, clients are unaware of the location or implementation of the objects they are using. The client of an object is a *domain*, which includes an address space, threads, and other resources. To invoke operations on the object, a client must have the *representation* of the object, which is simply the information needed to direct operations to the implementation of the object (see Figure 1). Ordinarily, clients are unaware of the details of the representation. An object implemented in another domain will usually have a representation containing one or more doors. A *door* is a capability–like handle[15] that is used for making protected and remote procedure calls to external object managers (usually called *servers*). We also allow lightweight objects, where both the state and the implementation of the object reside in the client domain.

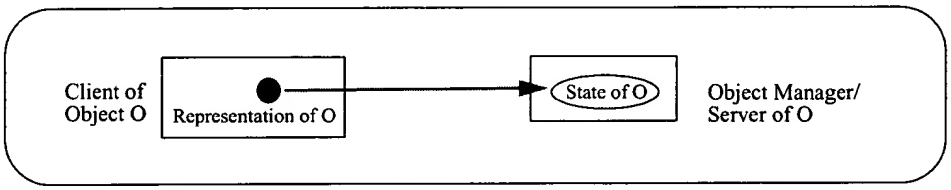

Figure 1. Client and Server of an Object

A door is implemented by the Spring microkernel, is valid for a particular domain, and is not forgeable. Doors form the basis for security in Spring; most Spring objects are capability–like unforgeable objects that use doors in their representation (such objects are as secure as doors are).

Doors are not persistent—a client cannot store a door in persistent memory for later use. When a client dies, all its doors become invalid. When an object manager dies, all its exported doors also become invalid. Although more complicated representations with more persistence have been implemented [14], the most common way of accomplishing persistence is to bind objects into the name service, and later reacquire them by name.

3 The Basic Spring Naming Model

If we are to succeed with a single uniform name service, the naming model needs to be general enough to capture all our naming needs. We use the model described in [1][19], which has been shown to describe a wide range of naming schemes

Names have meaning only in the context in which they are used. The name service allows an object to be associated with a name in some context. A name is said to *denote* an object. A *context* is an object that contains a set of name–to–object associations, or *name bindings*. In Spring, a context is any object that supports the context interface. An object may be bound to several different names in possibly several different contexts at the same time. In general, an object is unaware of the names that are bound to it. Indeed, an object may not have any names bound to it.

Resolving a name is an operation on a context to obtain the object denoted by the name. *Binding a name* is an operation on a context to associate a name with a particular object. These operations return or take as a parameter the object itself, not a lower–level identifier for the object, as some systems do[10][12].

A context is like any other object, and therefore can be bound to a name in some context. By binding contexts in other contexts we can create a *naming graph*, a directed graph with nodes and labeled edges, where the nodes with outgoing edges are contexts. Given a context in some naming graph, a sequence of names can be used to refer to an object relative to that context. Such a sequence of names is called a *compound*

name. A name with a single component is called a *simple name*. Informally, we refer to the naming graph spanned by a context as a *name space*, which includes all bindings of names and objects that are accessible directly or indirectly through that context.

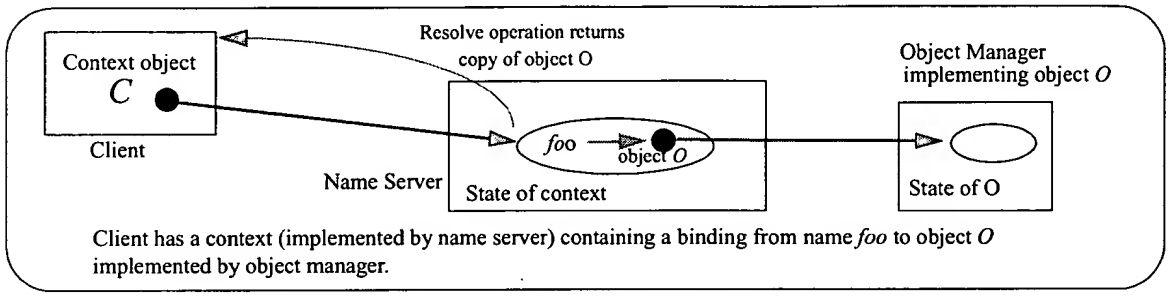

Figure 2. Separation of Client, Name Server, and Object Manager

A client of the name service performs naming operations via operations on context objects. The name service is implemented by one or more object managers, called *name servers,* which implement context objects. As shown in Figure 2, a client can use a context object implemented by a name server to acquire an object implemented by some object manager. Figure 3 shows a naming graph partitioned into three parts implemented by two name servers. In general, a client is unaware of which name server contains which parts of the name space, just as clients of any object in Spring are unaware of the exact implementation of the objects they use.

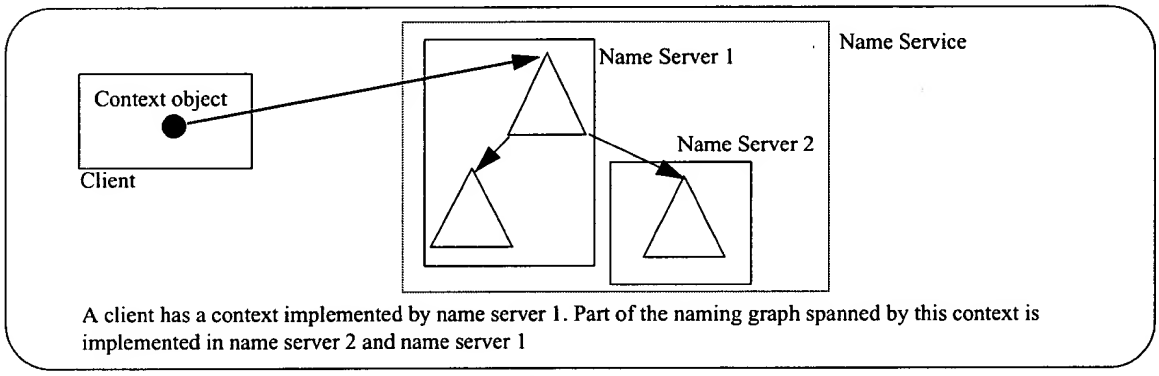

Figure 3. A Name Space Partitioned into 3 Parts Implemented by 2 Name Servers

In many systems, a program has two contexts: the root of some global name space, and the local context for the program. In Spring, a domain can have as many context objects as desired. A domain may be started with contexts given by its parent domain as implicit or explicit arguments, or can get contexts as a result of invoking or being invoked. In particular, resolving a name that denotes a context returns another context. A process can manage its contexts in various ways. They can be assigned to different variables in the programming language or bound to names in contexts. User interface programs may manage multiple contexts for the user; for example, a shell may allow multiple working contexts, or a desktop environment may associate them with folders on the desk. The meaning and usage of a context is determined by convention. If everyone believes that a particular context is "global," then it will have that characteristic. However, it is always possible to implement a context that is not connected into any other name space.

A context and the name space it spans can be manipulated directly. A context can be passed around like any other object. Name spaces can be composed by binding a context to a name in some name space. Given a set of contexts, one can construct a new context using that set, such as an ordered merge context (like union mounts in Plan 9), where a name space is created by merging the bindings from several contexts. The resulting new context can be used like any other context: it can be passed around, bound to a name, etc. Context constructions add flexibility without resorting to special name resolution rules or mechanisms. If we were not able to manipulate contexts directly, we might have been forced to limit the use of ordered merges to "mount" time (as in Plan 9), and to make the notion of merges a part of the naming interface.

3.1 Names

Names are conventionally represented as strings, are usually printable, and usually have some syntax for encoding different information by convention. To deal with issues of internationalization, and to be impartial with respect to accessing other existing name spaces, the Spring name service defines a structural representation for names, rather than encoding values such as version numbers and file extensions in the syntax. The presentation and parsing of names is relegated to user interface software.

A compound name is a sequence of components. Each component is an unordered set of elements. The identifier element of a component is an arbitrary UNICODE [7] string[4] that is never parsed (only compared for equality) by the name service. Other elements of a component encode version numbers (allowing references to the latest version), an object "kind" (analogous to file name extensions), etc.

Attributes add descriptive power to name components in a syntax independent way. In the absence of such a feature, users normally resort to syntactic convention. For example, in UNIX, suffixes such as *".c"*, *".o"*, etc., are used. Applications like the C compiler depend on such syntactic conventions to make name transformations such as from */foo.c* to */foo.o*. Unfortunately, such syntactic conventions are natural–language dependent. In Spring, we would like applications to make such name transformations for names in any character set. The *kind* attribute is used to address the needs of such applications. The *version* allows us to describe a specific version, without resorting to syntax like */foo!5*.

3.2 Naming Operations

The primary interface between a client and the name service is the context interface. There are issues of type safety that are beyond the scope of this paper, which are resolved using parameterized type interfaces [17]. The following are the common operations on contexts:

- named_object = context.resolve (name, mode)

  Returns the object denoted by the name. The mode argument indicates intended use of the object, and is described in more detail in Section 5.3.

- context.bind (name, binding_type, named_object, acl)

  Binds the name to the object in the context (or in a context reachable from it, if the name is a compound name). The binding type is used to distinguish bindings that the name service has to process during resolution: symbolic links specify *symbolic_binding*, name spaces are grafted by specifying *context_binding*, and normal objects specify *normal_binding*. The *acl* is the binding's access control list.

- new_context = context.bind_new_context (name, acl)

  Creates a new context with a particular name. It is also possible to create contexts that are (as yet) unnamed, although this is an operation on the name server interface, not the context interface.

---

4. Our current implementation simply uses ASCII strings.

- iterator = context.get_all_bindings (name)
  Returns the binding information in the context.
- context.unbind (name)
  Deletes a binding.

3.3 Naming Policies

The Spring name service does not prescribe particular naming polices. Our current policy is to provide a combination of system–based shared name spaces, per–user name spaces, and per–domain name spaces, as described in Section 8.2. Each domain is passed by default at least one context by its parent.

3.4 The Name Resolution Process

The process of name resolution (and, in part, any naming operation) starts with presentation of the name to the name service through a context object. If the name is a simple name, then that context performs the requested operation. However, if the name is a compound name, the first context obtains the object named by the first name component (which must be another context in order for the operation to proceed), then forwards the operation to that context, with the remaining part of the name as a parameter. It will often be the case that the second context object is implemented by the same name server as the first, in which case the forwarding may be implicit. Nonetheless, at any point it is possible for the next context to be in a different name server, requiring the request to actually be forwarded. The Spring name service defines an interface for name servers to use for this purpose.

In most cases, the request ultimately ends with an operation on the object manager—for example, to obtain the requested object on a resolve operation. This operation is through the name–sever to object–manager interface. The three interfaces—the client interface, the name–server to name–server interface, and the name–server to object–manager interface—define the architecture of the Spring name service. The rest of this paper will mostly focus on the latter two interfaces and how they are used to provide persistence and security for objects in Spring.

4 Persistence of Objects and Name Bindings

Figure 4 shows three different kinds of persistence that must be considered: (A) the persistence of the object state, (B) the persistence of access by a client to the object, and (C) the persistence of the binding between a name and an object.

The first kind, the *persistence of the object state*, is an issue for the object manager. It can use whatever facilities it chooses to store the state of the object for subsequent use. This information can be stored and retrieved at any time, although it is typically stored when no client has an object that refers to it, and is retrieved when a client is given access to the object. Another consideration is that one may want to trigger the object manager to make the object state persistent only if some client is interested in the object being persistent, so that if clients are interested in only transient forms of the object, then no state needs to be saved.

The second kind, *persistence of access to the object* (or a persistent reference to the object), is common in many object oriented systems, such as Clouds [12]. Values (called object ids) are typically used to designate objects. An object id is effectively a low–level persistent identifier that logically identifies the object, and is translated during object invocation into the object's physical location (since the object may move) by what is often called a location service. Although a client can store such object ids, it usually has several, and has to distinguish them by names; thus, such systems usually provide higher–level name services to map meaningful names to object ids.

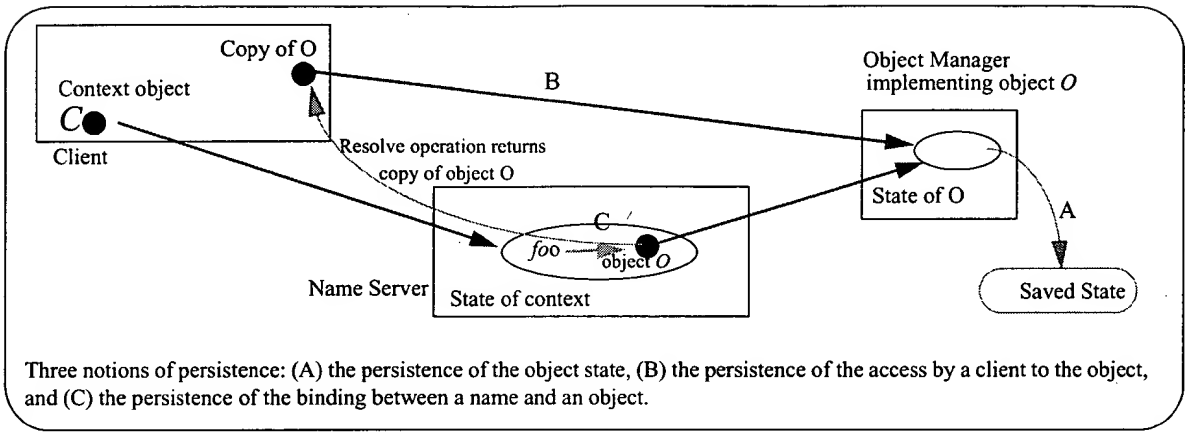

Figure 4. Notions of Persistence

Such systems use the object id for both persistence of access and for normal access during invocation. We decided to avoid this, and instead, separate these for two reasons. Firstly, there are many objects that will never be persistent, and therefore should not have to pay any cost to support general persistence. For example, domains are by default not persistent. Secondly, the representation of a Spring object—the part held by a client—is generally designed so as to quickly forward a method invocation to the server implementing the object. If the representation were some form of a persistent identifier, we would have to perform the translation step mentioned above. We believe this cost (or that of caching the translation) should not be necessary during object invocation. Clients generally deal with objects in their active form, and require the fast normal access to objects, and only occasionally deal with the persistent form of objects requiring persistence of access. If the two are separated, the translation step can be performed when some persistent reference to an object is turned to an active form.

In Spring, instead of using low–level persistent object ids for persistence of access, we use the name service and normal names. To support this, we provide the third kind of persistence, *persistence of name binding*. A Spring client that wishes to have persistence of access will bind the object to a name in a persistent name server, and later resolve the name when it needs to access the object. The object returned by the name server will contain a representation suitable for fast invocations and for passing the object efficiently as an argument.

Persistent name bindings may not be required of all name spaces, nor for all parts of a given name space. Some name servers simply hold name–to–object bindings in primary memory, implementing a transient name space that disappears when the server dies. Such policies are essential to avoid burdening transient objects with the overhead associated with persistence when accessed via the name service.

How does the name service make an object persistent? In the UNIX operating system, the file management and the name space management are implemented together, and it is easy for the name service to turn a file to and from its persistent form. Choices takes a similar approach by requiring that all object managers reside with the persistent name server. We find this approach impractical, since naming is a generic service for an open–ended collection of object types, and the name service is separate and autonomous from the servers implementing the various objects in the system.

When an object is bound in a persistent name server, the name server must contact the object manager, notifying it that one of its objects needs to be persistent, and must obtain appropriate information that will be needed later when the object is (re)created[5] at name resolution time. For this we have designed the freezing/melting interface. *Freezing* is the transformation of the representation of an object into a form

called a *freeze token*, which is a data value, not an object. That value can later be *melted* to generate an object. Thus, the name server stores name–to–object bindings as name–to–freeze–token bindings.

Since freeze tokens are forgeable, the connection to the object manager for freezing and melting requires authentication. The freezing service ensures that the same principal is performing the freeze and melt operations.

The freezing/melting interface can be used for all types of objects, can be used by any client (including the name service itself), and insulates the name service from the details of making the state of a particular kind of object persistent. It provides a uniform way of dealing with persistence, and defines the requirements for adding to the system new objects that wish to use the name service.

5 Security and Access Control

There are four issues that have influenced our security design: trust, protection, convenience, and performance. The first two relate to the integrity of the name service itself, and how that impacts the objects that use it; the last two are less fundamental from a security perspective, but are essential for the general usefulness of naming.

5.1 The Four Issues

The first issue is trust between the parties involved in naming. We do not assume that all naming components are trustworthy, but rather assume that trust is established when needed by means of authentication. Resolving a name can involve multiple name servers and an object manager, possibly crossing boundaries of trust. Figure 5 shows the parties involved in one example: the client, two name servers, and the object manager implementing the object bound to the name being resolved.

The client interacts with name server A when performing an operation on context C—say, resolving the name <*a b c*>.[6] Name server A does not trust the client, and will require authentication. Server A resolves the first two components <*a b*> to a context implemented in server B. Server B may or may not trust server A when it says that it is performing the operation on behalf of the client. Similarly, the object manager may or may not trust name server B when B requests the bound object on behalf of the client. Because we allow untrusted components to participate in naming, we must be able to establish trust where appropriate by means of authentication.

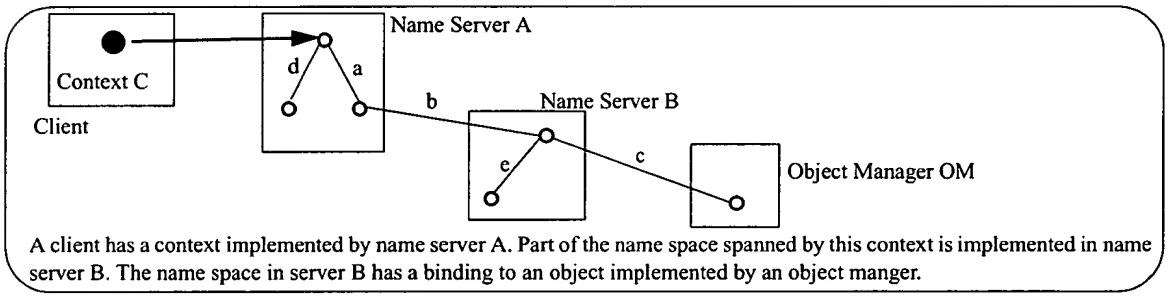

Figure 5. The Parties Involved in a Name Resolution

5. We don't consider the state of an object stored on a disk to be an object, and hence refer to creating an object from a stored state rather than activating an object.

6. We use the notation <a b c> in this paper; however, this is not the syntax for naming in Spring.

The second issue is protection of the name service information from unauthorized operations. A name server must protect itself by ensuring that clients (which may be other name servers) performing naming operations are permitted to do so. The name service must supply the means for clients to specify who may bind objects into, unbind objects from or otherwise alter its name space. Similarly, object managers also want control over which clients can access its objects and in which modes.

The third issue is to make it convenient for clients to get, and for object managers to provide, access to objects through the name service. If it is inconvenient, we would expect other services to arise that provide objects instead of the name service. In the typical case, a client resolving a name wishes to get an authenticated, ready–to–use version of the object. The object manager would like to receive a single operation with enough information to validate the request and produce the appropriate object. We would like to keep the protocols simple and the number of steps to a minimum while providing sufficient security. While some object managers may wish to implement their own access control lists, many others would prefer to leave this to a trusted name server through which their objects are made available to clients. Since almost every entity of interest in Spring is implemented as an object, relegating access control lists to name servers makes the general task of implementing objects simpler.

The last issue is performance. By developing and reusing trusted relationships for multiple operations, it is possible to reduce the cost of obtaining secure objects. In particular, the normal cases, where trust is well–known, can be efficient, and the expense of establishing trust can be borne by those who use different levels of trust. In addition, by returning an authenticated object to the client when possible, authentication overhead is reduced.

In Spring, the name service is the place where a client turns forgeable names into authenticated, capability–like objects. It is crucial that the name service perform name resolution in a secure fashion. During a name resolution, the name service must:

1. Check all context nodes in the resolution path to see if the resolution is permitted.
2. Check to see if the client is allowed to access the object in the mode he has requested.
3. Arrange to return an authenticated object in the requested mode.

5.2 Authentication

A Spring object is like a capability: a client with an object is ordinarily assumed to be able to perform the operations defined by the object.[7] A client usually uses an authenticated context where the principal on whose behalf naming operations are performed is encoded in the state of the context. The principal is most often established by an authentication process that is performed to get the initial contexts for a domain. From such authenticated contexts, a client resolves names to obtain additional objects without authentication steps unless and until it encounters an indirect name server or object manager that does not trust the name server that implements the context object on which the client performed the resolve operation.

In Figure 6, consider the client, who holds context C authenticated for principal P (the state of the context in name server A will encode the principal P), and resolves the name <a b c> to obtain the denoted object also authenticated for principal P. If server B trusts server A[8] when A says that it is performing the request on behalf of authenticated principal P, and if the object manager similarly trusts server B, then the resolve request can be performed without any authentication; the denoted object, authenticated for P, will be returned to the client. If B does not trust A, then the request will be interrupted (an exception will be

7. An object may be encoded with "any principal" or "untrusted principal", and may support only the authentication–related operations necessary to produce a new object with greater capabilities after authentication.

8. If server A and B are in the same address space, then they trust each other implicitly.

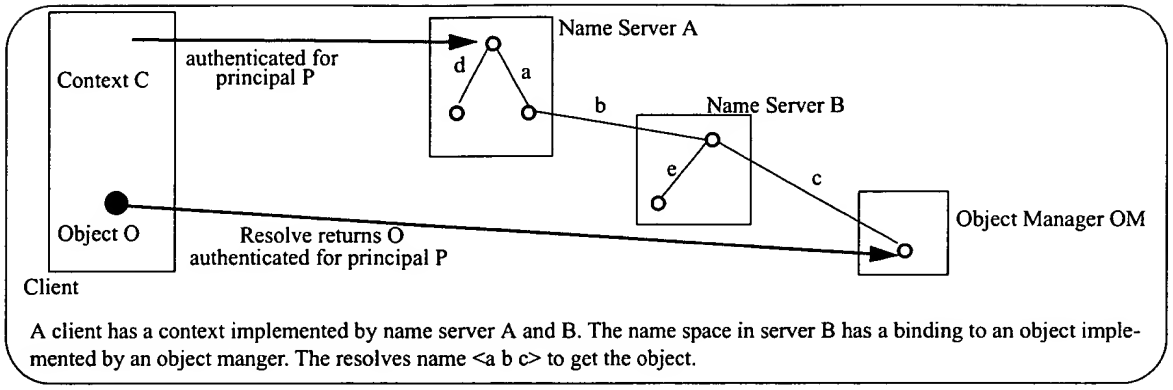

Figure 6. Authentication and Name Resolution

raised), which will cause the client to authenticate with B before continuing. Similarly, the client will have to authenticate with the object manager if the object manager does not trust B.

The trust between two name servers may be complete, nil, or partial. This trust is established and encoded in the authenticated context object that connects the name spaces implemented by the two name servers. If name server B trusts name server A completely, the connection object (link b in Figure 6) is a special, authenticated, name–server to name–server context which allows A to perform operations on behalf of *all* principals. The name–server to name–server context interface has the following operation:

named_object = resolve_on_behalf (name, mode, principal_name)

On the other hand, if a normal context object, authenticated for a particular principal P, connects name servers A and B, then A can perform operations on behalf of P only.

For example, the local file server (which implements the file name space)[9] completely trusts the local machine name server, and therefore a name resolution along a path from the machine name space to the file name space does not require additional authentication, since the two name spaces are connected via a name–server to name–server context. (Our naming environment is described in Section 7.6.) On the other hand, context objects bound in a domain's private name space are authenticated using the domain's principal, to avoid additional authentication.

The authenticated connections between two name servers and between a name server and an object manager are established once, and are reused to avoid authentication overhead. Similarly, the relationship between name servers and object managers is established using authenticated objects, as discussed in the next section.

Passing Contexts as Capabilities

An interesting case arises when a client passes a context to another client who is a different principal. As is the case with other objects, possession of the object ordinarily permits the second client to perform operations as if it were the first client. However, when a multi–component name is resolved in a context, the resolution path may cross into servers other than the one that implements the context. Thus, for contexts, the delegation of authority when a context is passed is valid only so long as trust boundaries are not crossed. At that point, the second client must either proceed under its own identity, or must negotiate through the first client for access to further contexts, just as the first client would do if it were issuing the request. Thus, an

9. Note that while name servers are generally separate from object managers, for access to our UNIX world and UNIX compatible disks, we have file servers that manage both the files and the file name space, and are accessed through the standard naming interface.

authenticated context is a capability that allows access to a *portion* of the name spaces spanned by that context without further authentication. This portion includes the name space managed by the name server implementing the authenticated context (e.g. the portion implemented by name server A in the case of context C) and recursively the portions implemented by name servers that trust this name server (e.g., the portion implemented by name server B if B trusts A).

5.3 Access Control on Naming Operations and Objects

The name service defines and implements an access control model based on Access Control Lists (ACLs). Each context has an ACL; a principal is granted a set of rights encoded in a bit mask, for example, to permit searching, binding into, or deleting from contexts. Since this facility was required to control operations on context objects, we decided to provide an ACL on each binding for access control of objects. An object manager may implement ACLs for its object if it desires, although many object managers export their objects through name servers they trust, and rely on the per–binding ACLs provided by these name servers.

This raises an interesting issue: the access control depends on the path used to resolve names. Note that even in traditional name services that allow DAGs (as hard or symbolic links), the access control depends on the path used to resolve a name, with the final access control being on the object itself. The same is true for our scheme when object managers implement their ACLs. However, some object managers do not implement any ACLs, and simply rely on the per–binding ACL. Such object managers usually publish their objects through only one name server (the one they trust), and the per–binding ACL where the object manager bound the object can be viewed as "the object's ACL." Of course, a client who obtains such an object can always rebind it in a different part of the name space to share with others, but this object will retain the limited access with which it was originally issued.

Because the access rights for contexts cannot anticipate all the possible semantics of objects, we introduce the notion of *mode*, which is a value that defines the set of rights with which the object is intended to be used. The name service checks that the principal (as encoded in the context object) making the resolve request has the rights specified in the mode parameter. It passes the mode parameter to the object manager, in order to permit it to create an object with no more access rights than specified in the mode—for example, to allow a client with read and write access to obtain an object that permits only read access. The resolve operation fails if the client is not entitled to access the object in that mode.

The meanings of the access rights are determined by the semantics of the objects. Generally, they are considered part of the interface for that object. There is one mode, called the *default mode,* that is understood by name servers and all object managers. When this mode bit is specified to the resolve operation, the object will be returned with the maximum possible rights for the given client.

5.4 Summary

The Spring name service allows untrusted components to participate in providing naming and publishing objects. Authentication is done at appropriate times to establish trust. These trust relationships are encoded in capability–like authenticated context objects, which are reused to avoid authentication complexity and overhead after trust is established.

Since the name service needs to implement authentication and access control to protect itself, it is natural to have the name service provide those same services in an integrated way for the convenience of clients and object managers. When trust cannot be established, or when object managers have stronger security requirements, they resort to direct authentication between the client and name server or object manager.

6 Interactions between the Client, Name Servers, and Object Manager

The preceding sections have discussed the requirements and general approach for persistence and security. This section shows how the three participants in naming—the client, the name server, and the object manager—interact. The interfaces and protocols provide flexible support for persistence and security, and allow a variety of policies, implementations, and optimizations.

6.1 Client and Name Server Interaction

During a resolve operation (and, in fact, any operation with a compound name, since a resolution occurs to navigate the naming graph), if the name server gets to a point where it cannot continue the resolution, but where the client may be able to, it will raise the *cannot_proceed* exception. The exception returns information needed to continue the operation, such as the name processed so far, the rest of the name to process, a context in which to continue the naming operation, and other data required to complete the naming operation. The exception is raised in various situations, including inadequate implementation, handling of symbolic links, and when boundaries of trust are crossed during the naming operation.

A client–side name service library catches the exception and continues the operation. A resolve operation is processed in the client side library as:

```
while {
    try {
        result = context.resolve(name, mode)
        exit_loop
    } except ex {
        cannot_proceed {
            if (ex.reason == cannot_continue_resolution) {
                context = authenticate_if_necessary(ex.context_in_which_to_continue, credentials);
                name = ex.reminding_name
            } else if (ex.reason == cannot_authenticate_object) {
                result = authenticate_and_get_object(ex.zero_rights_object, credentials, mode);
                exit_loop
            } else ...
        }
    }
}
```

When a trust boundary has been crossed and an exception is raised, it is necessary for the client to authenticate itself with the returned context or object in order to continue the operation. Note that this is also the opportunity for the client to authenticate the new name server, since the client is trusting the name service to provide the proper objects (in order to avoid Trojan horses).

6.2 Name Server and Object Manager Interaction

After the name has been resolved to the point of a particular binding, and after access rights have been checked to ensure that the operation is permitted, the name server needs to interact with an object manager. A persistent name server interacts with an object manager to freeze and melt objects. This is done through the object manager's *freezing_service*. A transient name service, or a persistent name service that has melted once and cached the resulting object, interacts with the object manager during a resolve operation to generate a copy of the denoted object in a specific mode authenticated for a particular client. This is done through the object manager's *duplication_service*.

Both the duplication and the freezing service objects may be implemented directly by the object manager or delegated to some other service. Generic forms of these services are available for simple object manag-

ers. Name servers typically cache authenticated connections to the duplication and freezing services, but of course, they may be revoked by the object manager or deleted and reacquired by the name server at any time. Also, as we show below, duplication, freezing, and melting may in some cases take place locally at the name server.

The name service obtains these services using the *object manager identifier,* which is simply a name that each object provides via the get_object_manager_id operation. The name service resolves the object manager identifier in a well–known context, resulting in an appropriately authenticated freezing or duplication service object. Obtaining the services indirectly via a name instead of directly via the object is an important factor in freezing and melting securely (as explained below).

6.2.1 Duplicating an object

The object returned as the result of a name resolution usually encodes the appropriate principal and the requested mode. The *dup* operation on the duplication_service takes the object to be duplicated, the desired principal, and the desired mode, and returns an object authenticated as that principal and enabled for that mode. If the client requested the *default mode*, then the maximum allowed mode is specified to the dup operation.

Most object managers make their objects available to their clients by binding these objects in some name space. If the object manager does not trust the name server managing the name space through which it publishes its objects, then it must implement its own ACLs. When a client resolves the name of such a published object, a duplication request arrives at the object manager. If the object manager implements its own ACLs, it will check it and raise an *access_denied* exception if access is not permitted. If the object manager trusts the name server, then it accepts its word on the principal, and returns an authenticated object in the requested mode. If it does not trust the name server, then it will return an unauthenticated object, and the name server will propagate the result to the client via a *cannot_proceed* exception, allowing the client to perform an authentication step.

A client will often bind an object that was obtained in a specific mode for a particular principal into a name space for sharing with another client. When a duplication request arrives for such an object, the object manager will compare the principal requested with that encoded in the object; if they are the same, then it will return a duplicate object with a mode that is the same as or less than the original mode (note that since the principals are the same, it not necessary for the object manager to trust the name server). If the principals are different, the object manager implements its own ACLs, and the ACL permits the requested mode for the requested principal, then an appropriate authenticated or unauthenticated object is returned, depending on whether the object manager trusts the name server. In all other cases, it will raise the access denied exception. What is interesting here is that when an object is passed as a capability via the name service, it will succeed only if the principals are the same or if the object manager implements its own ACLs.[10]

If there is no duplication service, then the object manager does not support modes or encode principals, and the object representation can be copied locally by the name server. There are also some degenerate duplication services implemented by name servers—for example, to copy the state of an object that is only data. Exotic duplication services are also possible—for example, a duplication service might limit the number of copies of an object that could be allowed at once.

6.2.2 Freezing and melting an object

The freeze and melt operations are performed as part of binding and resolving names in a persistent name server. In both cases, it is necessary for the name server to interact with the freezing service *without using the object.* When melting, the object is not available, only its freeze token is, and the name server must

10. We are currently investigating alternative ways of dealing with such capabilities bound in a name space.

convince the freezing service that it is the same name server that froze the object previously. This is one reason why during freezing, although the object is available, freezing is not an operation on the object, since the object manager needs to establish who is asking for the object to be frozen.

A second reason is that it is important that the name server ensure that it freezes and melts objects at the same freezing service (or rather, obtains the freezing service in the same way). Otherwise, it is quite easy to compromise the security of the name service. For example, one may consider it natural to combine the steps to get the object manager identifier (omid) and the freeze operation as part of a single (freeze) operation on the object. Then one could implement bind and resolve as:

```
bind(n, bt, o):

    <omid, ftoken> = o–>freeze();
    store_binding(n, bt, omid, ftoken)
```

```
resolve(n, mode):

    <omid, ftoken> = get_binding_for_name(n)
    fs = get_authenticated_freeze_service(omid);
    o = fs–>melt(ftoken, mode);
    return o
```

This approach has a serious security problem. A malicious client in concert with a malicious object manager can trick a name server to give, for example, the password object in write mode as follows: the malicious client binds a name to an object implemented by the malicious object manager. During the freeze operation, the malicious object manager returns to the name server the object manager identifier of the password object (which can be obtained from the password object) and the freeze token of the password object (which is forgeable and it has been able to guess). The malicious client resolves the object for write–mode (it has permission, since it did the bind). The name server is tricked into performing the melt operation at the freezing service of the password object and returning the object in write–mode.

Our design ensures that freezing and melting are performed at the same place:

```
bind(n, bt, o):

    omid = o–>object_manager_id()
    fs = freezing_ctx–>resolve(omid)
    ftoken = fs–>freeze(o)
    store_binding(n, bt, omid, ftoken)
```

```
resolve(n, mode):

    <omid, ftoken> = get_binding_for_name(n)
    fs = freezing_ctx–>resolve(omid)
    o = fs–>melt(ftoken, mode)
    return o
```

Since freeze tokens are forgeable, connection to a freezing service must be authenticated in order to ensure that the name service is trusted to melt the particular object. For practical purposes, freeze tokens are not globally unique but are scoped relative to the freezing service that issued the token.

During the life of a system, object managers and their freezing services will occasionally crash. One could require system administration to arrange for the object managers to be restarted after every crash. A better approach would be for the melt operation to trigger the start of the corresponding object manager. This would allow object mangers to be up only when active instances of their objects are around, and die otherwise. This can be made to occur easily using the freezing scheme recursively. Before melting an object, the name service first tries to access the freezing service by resolving its object manager identifier. This requires melting the freezing service object, and can trigger the object manager to be restarted if it is not running. For bootstrapping, we require freezing services to be frozen and melted at special services; the system only has to ensure that such services are kept alive.

As with duplication services, freezing and melting are usually done by the object manager, but may be done elsewhere. One special case freezing service is used for objects that are only data. In this case, the

freezing service simply returns the representation of the object as the freeze token; the melt operation regenerates the object from the token which contains the representation. This generic freezing service is available in libraries that are linked into name servers.

Another interesting case is the "transient object freezing service," which can be used for objects that cannot be made persistent (such as the object that represents a domain). This service simply keeps the objects passed for freezing in primary memory and issues freeze tokens, subsequently returning the objects when asked to melt them. This service may conveniently be linked into various object managers, or can be implemented separately. It relieves all persistent name servers from handling unfreezable objects, and provides the uniformity where all objects, even those that cannot be made persistent, can be bound in persistent name spaces.

6.3 Name–Server to Name–Server Interaction

A multi–component name resolution (on a resolve or other operation) can cross name server boundaries. Consider a name resolution crossing from name server A to name server B in Figure 6. Server A performs part of the resolution, and forwards the remaining name to server B, giving the name of the principal on whose behalf the resolution is being performed. Server B returns a properly authenticated object or raises an exception, such as *cannot_proceed*. Server A returns the results to the client.

Name server B raises the cannot_proceed exception in two situations. One is when this exception is raised by an object manager or another name server with whom name server B interacts during the resolution; in this case, name server B simply re–raises this same exception. The second situation arises when server B does not trust A, in which case server B will raise the *cannot_proceed* exception, requiring the client to authenticate itself with server B and continue the operation.

As we mentioned in Section 5.2, two name spaces implemented by two name servers are connected by an authenticated normal context object or by an authenticated name–server to name–server context object.

7 Use of Name Services in Spring

This section illustrates how the name service is used in Spring for a wide variety of purposes. It demonstrates the general usefulness of the context interface, and the range of implementations that are possible.

7.1 Uniform Name Services

Although Spring does not have the concept of a global name space, the name service is universal, in that any object to be named can be named using the same name service: this applies whether the object is local to a process, local to a machine, or resident elsewhere on the network; whether it is transient or persistent; whether it is a standard system object, a process environment object, or a user specific object. (Note that in Spring, not all objects are named. However, any object that is to be named can indeed be named using our name service.)

As a result, tools to browse and manipulate name spaces can be used on any name space, whether for debugging a transient collection of objects, or for searching for file objects. Although not all name spaces are linked together, over time we have composed different name spaces in different ways. For example, the name space given to a domain when it is created is composed of transient and persistent, and of local and networked name spaces.

7.2 Ordered Merges

Ordered merges in other systems are usually restricted to special situations, such as command search paths or unions at mount time. In Spring, since context objects can be manipulated directly, we did not clutter the

context interface with a notion of merges. Instead, we provided merges as a separate contexts implemented in a library. We give several examples below of our use of ordered merges in constructing name spaces.

7.3 Generic Implementations

Generic implementations of contexts are available in dynamically linked libraries. They provide persistent and transient name spaces, and implement the full security model. We also have a client side name service library that handles the *cannot_proceed* exception and other client side responsibilities. The name space of a domain, the name space of a machine, and the name space shared by a set of machines all use these generic implementations. Applications and services whose primary purpose is not naming use a much simpler and more specific implementation.

Currently, we have several freezing services. For example, we have a file server that manages unnamed files; its freezing service returns an extended inode number as a freeze token. Data–like objects use the generic freezing service. By default, objects that do not have their own freezing service (because they are not persistent) use the transient object freezing service mentioned in Section 6.2; as a result, all Spring objects can be bound in a persistent name space.

7.4 Interoperation with Foreign Name Spaces

Sun's NIS® directory provides information about various resources in our UNIX network. We have implemented a gateway that allows access to the NIS name space from Spring. The gateway supports the context interface, and translates between NIS names and Spring names, handling issues of syntax. Thus, Spring clients can set or retrieve NIS data using the Spring naming interface.

7.5 Accessing Foreign Named Objects

We access files in the UNIX file system via a service that encapsulates the UNIX file system. We map the UNIX name space into a Spring name space, exporting UNIX files as Spring files and UNIX directories as Spring contexts. A client wishing to open a file performs a resolve operation on a context, specifying the access mode desired. The server forwards the request to the UNIX file system, and encapsulates the resulting UNIX file as a Spring object. In the encapsulation, we have extended the original UNIX file system semantics to allow arbitrary objects to be bound into contexts. For example, non–local files and non–file objects are simply frozen, and the resulting freeze token is stored in a file.

7.6 Policies: The Spring Naming Environment

The Spring name service does not prescribe particular naming polices. Our current policy is to provide a combination of system–based shared name spaces, per–user name spaces, and per–domain name spaces that can be customized by attaching name spaces from different parts of the distributed environment. We use ordered merge constructions for further flexibility in tailoring name spaces.

In Spring, machines are grouped into villages, based on geographic location, security considerations, administrative convenience, etc. There is a high degree of sharing within a village, but limited and controlled sharing across villages. A machine has a local name space for its local entities, and a village has a name space for the entities shared among machines. The village name spaces are connected via an enterprise level name space. These system based name spaces are used to make system resources available to clients.

Each user has a private name space that persists across login sessions, much like the home directory in UNIX. The main difference in Spring is that a user is allowed to attach various name spaces to configure a private naming view.

Each domain also has its own private name space that can be customized to access arbitrary name spaces—in particular, parts of the machine and village name spaces, and typically, the per–user name space of the user on whose behalf the domain is executing. The per–domain name space also incorporates private name spaces such as the environment variables name space. The per–domain view captures the dynamic and run–time environment of a computation. In particular, it has bindings for system objects such as binaries and libraries based on the current site of execution; as remote execution is performed or as a process migrates, the per–domain view may change. In general, it is not necessary for a domain's naming view to persist when the corresponding domain dies. A domain's private name space typically contains the following (see also Figure 8):

- Private name bindings

  The domain's name space has bindings for environment variables and program input/output objects. (A special mechanism to pass standard I/O objects as implicit parameters is not needed.)

- Shared name spaces

  Several shared name spaces are attached to the domain's name space using well–known names: ~ (the per–user name space of the user owning the domain), *user* (the name spaces of different users), and *dev* (devices). If there were, for example, a worldwide global name space, we would attach it to the name space of a domain using a well–known name. The use of well–known names provides coherence (transparency) in naming in the distributed environment.

- Generic name spaces containing standard system objects

  A domain's name space has generic name spaces that contain system objects: *sys* (executables under *sys/bin* and libraries under *sys/lib*) and *services* (such as *services/authentication*). The name spaces of *sys* and *services* have parallel structures in both the machine and village. These name spaces contain copies of such system objects for local use.[11]

  The *sys* and *services* name spaces in a domain's private name space are typically an ordered merge of the corresponding name spaces of the machine and village. The merge arranges for the local instance to be visible first. A user can also keep similar structures in his home contexts, which can be used for further tailoring of these name spaces. During remote execution, the merges are automatically re–evaluated to take advantage of the name spaces of the remote site.

Each parent domain passes to its child a context defining the child's private name spaces. This is usually a copy of the parent's context. Since most domains make few changes to their name spaces, we make the child's context a copy–on–write copy of the parent's. Note that a domain can be given and can acquire other contexts. The init domain initializes a domain name space that it passes to its children; they, in particular the domains that are created as a user logs in, further customize their inherited naming view.

Implementation of the Naming Environment

The naming environment is implemented by several name servers. The machine/domain name server (one per machine) implements the machine name space and also the name spaces of each of the domains on that machine. The village name server (one per village, though it may be replicated) implements the village name space shared by all machines in the village. The domain name space has parts of the machine and village name spaces attached to it. (Figure 8 shows a typical configuration of name spaces and name servers.) The machine name space also has parts of the village name space attached to it. In addition, for access to our UNIX world and UNIX compatible file system disks, we have several file servers that implement the

11. The local copies are in addition to what is kept automatically by our caching scheme. Local copies allow a machine to maintain local autonomy, especially during disconnected operations.

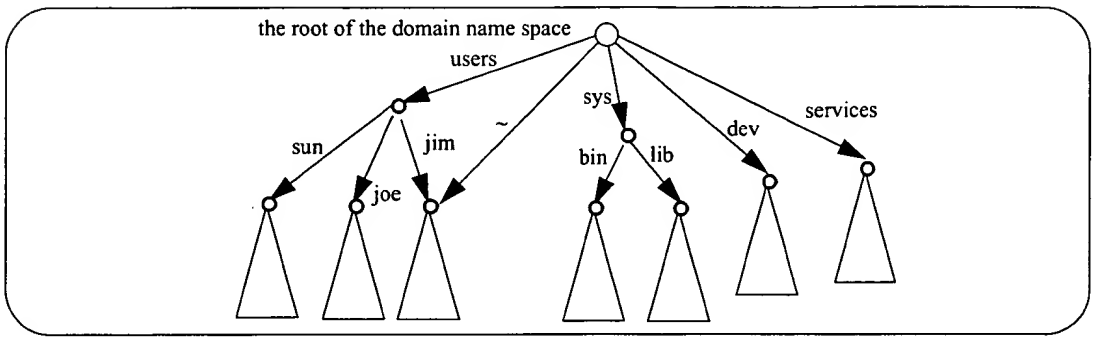

Figure 7. A Typical Per–Domain Name Space

files and the file name space accessed via our standard name service interface. These file name spaces are attached to the machine and village name spaces.

The trust between name servers is established and encoded in the authenticated context object that connects the corresponding name spaces implemented by the name servers. For example, in Figure 8, consider the name space of files on the local disk which is attached to the local machine's name space; the file server completely trusts the local machine name server, and therefore a name resolution along a path from the machine name space to the file name space does not require additional authentication. Similarly, the file servers in the village trust the village name server. The name–server to name–server context is used to attach the file name spaces to both cases.

On the other hand, context objects bound in a domain's private name space (see Figure 8) are normal context objects authenticated using the domain's principal, so that names resolved through these contexts do not require additional authentication as name server boundaries are crossed. This optimization significantly

reduces the authentication overhead when a domain resolves names. The village name server does not trust most machine name servers, and hence requires clients to authenticate on naming operations.

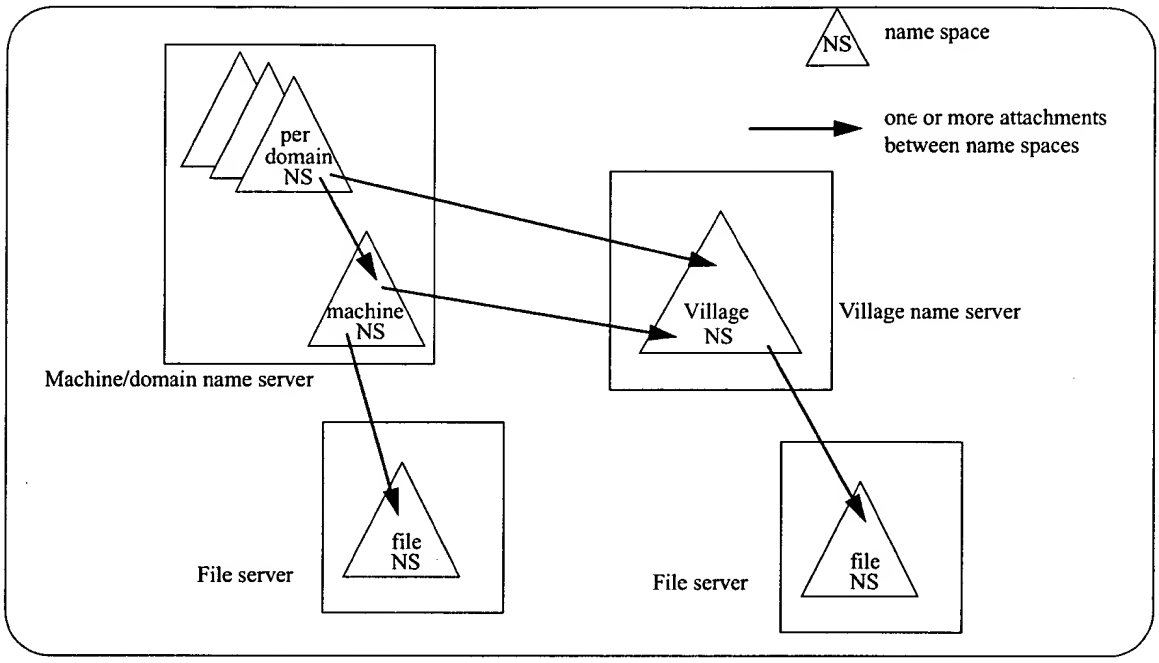

Figure 8. A Typical Configuration of Name Spaces and Name Servers

7.7 Contexts Implemented by other Services

Many services and applications allow access to various pieces of information distinguished by identifiers. The context interface is useful to such subsystems, even though their primary business is not the name service. One example is that of the generic monitoring objects used by many different services to give statistics about the server. The various values, represented as data–only objects, are accessed by names through the context interface. The implementation needs to support only two operations, *resolve* and *get_all_bindings*, much as any read–only context would.

A second example is in the dynamic object invocation service, which allows an invocation to be performed without first having been compiled. Using interface specifications from a database, arguments to the operation are bound in a context object, using IDL formal parameter identifiers as the names. The context is passed with other information to the invocation service, which marshals the arguments, performs the call, and returns the results in a similar context.

7.8 Virtual Worlds and Name Spaces

One often needs to build virtual worlds, in which clients are insulated from aspects of the real world. This is difficult to support with traditional name services that have rigid notions of name spaces and root contexts. In Spring, it is easy to create a parallel set of name spaces for particular purposes. In our development environment, several releases of the system are tested and used simultaneously. Each new release of Spring publishes executables, libraries, and services, which are accessed through the release versions of the *sys* and *services* name spaces. In most cases, developers do not break binary compatibility, and need to override only some objects.They use an ordered merge to overlay their name spaces on the standard release

name spaces, resulting in a parallel world with partly shared name spaces. When binary compatibility is broken, they create a private, separate world with its name spaces.

8 Relat d Work

8.1 Uniformity

As mentioned in the introduction, the lack of uniformity in conventional systems causes three problems. Firstly, clients have to use a different name service depending on the kind of object being named. Secondly, defining new objects or services that are accessed by name can be complicated and cumbersome, since the new objects must be made to "fit in" somewhere, or a new name space must be defined and implemented. Thirdly, the non–uniform name spaces generally cannot be composed together.

DCE[12] has made an attempt to improve uniformity by composing name spaces such as file system name spaces onto the higher level directory name space. This is done through special entities called junctions. For example, a junction point in the directory level name space called */fs* signals that one is entering the file system name space (it is like a specialized mount point). This provides network–wide access to files: a client can name files and file systems in other parts of the network. However, the client is still faced with multiple name spaces and their corresponding naming interfaces. (Furthermore, it not clear how easy it is to create new junctions when introducing new object types and their corresponding name spaces.) While DCE does not provide a uniform name service, it provides a uniform way of accessing the various name spaces in the network.

Plan 9

Given the lack of some unifying concept like an object, the Plan 9 approach is probably the best way of providing uniformity of naming in UNIX or other non–object oriented systems. However, in an object oriented system, our approach has several advantages over Plan 9's approach of attempting to make all objects look like files.

The first advantage is that in Spring, nameable objects have strong well–defined interfaces that are appropriate for the particular type of object. We do not attempt to make all objects obey the file interface and hide the object's true interface inside the read, write, and ioctl operations.

The second advantage is that the effort required to make new object types nameable is low. Nameable objects in Spring do not have to be made to obey the file interface, and object managers do not have to implement the name service. For example, objects implemented by the Spring kernel, such as domains (processes), VM objects, and kernel monitoring objects, are routinely bound into the machine's name spaces; the kernel did not have to change the nature of these objects or implement a name service to make this possible. This means that new nameable objects and new implementations of nameable objects can be added to the system at any time without modifying the implementation of the name service. Spring developers routinely create new objects and make them available via the name service.

In contrast, Plan 9 requires that nameable non–file objects be made to look like files, and that implementations of the non–file objects implement the (file) name service. Although Plan 9 has been able to make many objects look like files, it does require a certain amount of effort which is not required in our system. Furthermore, we believe that the Plan 9 approach cannot succeed for all kinds of objects. Indeed, the network name space in Plan 9 (the name space that binds file servers) does not map very easily to a file system, and hence is a separate non–file name service[5].

The third advantage is that Spring name spaces need not be segregated by the type of objects being bound in them. For example, we can store a file in our equivalent of the */proc* file system, and we can store a domain object in the file system. This functionality is not possible in Plan 9, since the */proc* implementa-

tion deals exclusively with processes being made to look like files, and the regular file system deals only with real files.[12] Thus, in spite of making most objects look like files, a lack of uniformity is evident, since each "file server" can only bind objects of the same true type.

Other Object Oriented Systems

Although, the uniformity of objects in object oriented systems offers an opportunity for a uniform name service, existing object oriented systems either have not provided a uniform name service or have provided one with serious restrictions. Programming language based object oriented systems such as Smalltalk rely on the name space of variables with nested scopes provided by the programming language. In Emerald, objects were generally obtained using the language based variable naming scheme, and did not use a general name service[20,21]. Others like Choices[10,11] have the serious restriction that the object managers must be integrated and reside with the name service; this makes it difficult to add new object types. Furthermore, Choices provides two name services: one for persistent objects and one for transient objects.

One of the main difficulties has been that the name service often needs to interact with the object managers for the purpose of persistence and security. For example, Choices avoided the problem by providing two name services, one for persistent objects and the other for transient objects. It also requires that the persistent name server and all the object managers of objects being named reside together, which makes adding new objects or implementations difficult.

8.2 The Naming Model and Environment

We chose a general naming model and made contexts and name spaces first class entities that can be manipulated directly and passed around like any other object. This first class nature is not merely a consequence of being object oriented—we could have simply required that names be resolved relatively through the root or working context. In UNIX, for example, opening a directory returns a file that can be used for listing the directory entries, but cannot be used for other directory operations, such as creating new directories. Also, UNIX applications that need to exchange and share a private name space would have had to build their own naming facility or incorporate the private name space into a larger system–wide name space and access it indirectly via the root or working context. Spring applications can share and exchange private contexts and name spaces. The first class nature of contexts has also made it very simple for us to support context constructs like ordered merges without limiting the use of ordered merges to "mount" time (as in Plan 9) and without making the notion of merges a part of the naming interface.

Our name service leaves policies to a higher level. Our current policy is based on per–domain (like Plan 9), per–user, and system based name spaces. We rely on attaching name spaces and ordered merges to tailor a per–domain name space. We have avoided Plan 9's per–process mount tables, which are clearly more flexible, for two reasons. Firstly, we found that ordered merges allow a process sufficient flexibility in configuring its name space, and have not found the need to add the notion of per–process mount tables. Secondly, when a private name space overlays a shared name space, resolving names in the shared name space becomes more complex in a distributed environment: one cannot blindly resolve path names in a remote shared name space, as private local mounts may overlay parts of it.

8.3 Persistence

In Clouds, all object references are persistent and the name server does not have to deal with the issue of making objects persistent. Spring object references, on the other hand, are not persistent. The freezing architecture is used to convert object references to and from their persistent form and also to communicate

12. The best one can do in Plan 9 is to use the union mount to attach special file systems behind the regular directory, so as to allow regular files to be created "ahead" of the mounted special file system.

to the object manager that one of its objects needs to be made persistent. The Spring name service plays a crucial role in persistence—a client that needs an object to be persistent simply binds the object in a persistent name space and later retrieves it using its name.

Choices provides two different name services, these being for persistent and transient objects, respectively. Spring provides a single name service interface for persistent and transient name spaces, and allows persistent or transient objects to be bound in either kind of name space.

In type specific name services such as the UNIX file system, the object manager and the name service reside together, and can easily turn objects to and from their live form; they do not need a general freezing architecture. Directory services avoid the issue by simply mapping names to data values, leaving the client to turn a data value into a live usable object.

8.4 Security

The trust between Spring name servers may be complete, nil, or partial, and is encoded in the authenticated context object that connects name spaces. Previous name services have used a more static structure for the trust relationships between name servers.

In the UNIX file system, trust between different parts of the file name space connected via the mount operation is not an issue, since all the name spaces are managed with the kernel. With NFS® (Sun's distributed computing file system), when a name resolution is forwarded to an NFS server, the server trusts the client machine as long as the principal (user id in UNIX) is other than the *root*.

In X500[6] , the directory service agents authenticate with each other, and when a name resolution along with the client's principal name is forwarded from one directory service agent to another, the second one trusts the first one to have authenticated the client.

In DCE, within a cell, name servers trust other name servers. Across cells there is no trust, and the client must have an account on the foreign cell and perform authentication. In Spring, even within a village (similar to a cell), there are different degrees of trust. For example (see Figure 8), the machine name servers trust the village name servers, but the reverse is generally not true. Within a machine, all name servers, including file servers, trust the machine name server. Furthermore, partial trust is established for name spaces bound in the per–domain namespace by using contexts authenticated for the domain's principal. In DCE, the equivalent of the *cannot_proceed* exception is handled in the client agent process, while in Spring, it is handled in a library in the client itself.

Spring architecture allows object managers to determine how much to trust a particular name server, and an object manager is permitted to forego the convenience of per–binding ACLs and implement its own access control and authentication, if it wishes. The issue of trust between object managers and name servers is fairly unique to our name service. It does not arise in conventional type specific name services such as the file system, because the name service and the object manager are implemented in the same server. Directory services avoid the issue by simply mapping names to data values, leaving the client to turn a data value into a connection to an object manager after authentication.

9 Concluding Remarks

The Spring system, including its name service, is in daily use for its own development and further experimentation. The name service provides acceptable performance in its various uses, although optimization work is ongoing. The system also supports UNIX compatibility mode binaries (the emulation package is a thin veneer on Spring services[16]).

The distinguishing features of the Spring name service are as follows:

- *Uniform and extensible name service.* Name services and name spaces do not need to be segregated by the type of the object—in principle, any object could be bound to any name.The name service provides critical support to integrate new kinds of objects and new implementations of existing objects into Spring. It also provides this same capability for name servers, including the ability to extend the name space and compose new name spaces.
- *General naming model with first class name spaces.* We use a general naming model, based on names and contexts, that is powerful enough to meet the various naming needs of the operating system, applications, and users. Contexts are "first class" objects that can be directly used to create and manipulate name spaces and to pass them around like other objects. A name space need not be part of some larger global name space; policies such as notions of root or working context are not part of the name service, but can be built on top of it. The naming interface is generic, and is applicable to non–traditional use, such as in spreadsheets.
- *Integrated support for access control and persistence.* The name service architecture provides access control and persistence for those object managers who wish to delegate those functions to it, while allowing them to retain control when needed. Clients see the name service and object managers well integrated, as in type specific name services.

While our object oriented environment made uniformity possible, there were a number of challenges, especially with regards to integrating persistence and security.

The Spring name service plays an important role in accomplishing the goals of Spring. We have successfully integrated all name–to–object mechanisms in one service. By masking the differences between objects of different types or of different implementations, we can add new facilities to the system, or can replace existing facilities in a first class way. Programmers and users do not need to learn new models and new methods of access, and are insulated from many changes and extensions.

In making Spring secure and robust, the overhead in complexity and development can be a substantial impediment. Having the name service provide access control and a framework for persistence has made it easier to have these capabilities pervasively in Spring services.

Other object oriented systems have made use of the uniformity of objects. Spring carries those objectives further, truly separating naming into a service in its own right, rather than making it an adjunct to one or more objects. Moreover, that name service is not just a generic addition to a collection of type specific name services, but is *the only* name service used in Spring.

The context interface has proven useful in a growing variety of situations in Spring. As we expand the application base and usages of the system, we find more ways to use the name interface, and discover more advantages of having access to functionality through a naming–style interface.

A result that remains to be proven, but toward which work is ongoing, is that the uniformity and extensibility provided by objects and naming at the system level will accrue benefits to users of the system.

10 References

[1] J. H. Saltzer. Naming and Binding Objects. In *Operating Systems: An Advanced Course*, volume 60, pages 99—208. Springer–Verlag, New York, 1978.

[2] Lampson, B., "Designing a Global Name Service", Proceedings of the 5th ACM Symposium on Principles of Distributed Computing, 1–10, Calgary Canada, Aug. 1986.

[3] Birrell et. al. "Grapevine: An Exercise in Distributed Computing", Comm. of ACM, April, 982, pages 260–273

[4] Pike R., Presotto D., Thompson, K., and Trickey H., "Plan 9 from Bell Labs" Proceedings of 1990 UKUUG Conference, July, 1990

[5] Pike R., Presotto D., Thompson, K., and Trickey H., "Use of Name Spaces in Plan 9", Unpublished report, 1992

[6] CCITT, X500, 1988

[7] UNICODE

[8] Thompson K., and Ritchie D.M, "The UNIX Timesharing System", Comm. of ACM, July, 1974, pages 365–375

[9] Cheriton D. and Mann T.P, "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance" ACM Trans. on Computer Systems, 7, 2 (May, 1989), pages 147–183

[10] Campbell R.H., Islam N., and Madany P., "Choices, Frameworks, and Refinement," USENIX Computing Systems. 5, 3 (summer 1992), Pages 217–257

[11] Madany P., "Personal Communication,", 1992

[12] J. M. Berabeu–Auban et al, "The Architecture of Ra: A Kernel for Clouds", *22th Hawaii International Conference on System Sciences*, Jan 1989, pages 936–945.

[13] Radia S. Madany. P., Powell M. P, "Persistence in an Extensible System with General Naming", internal Spring document

[14] Hamilton G., Powell M. P., Mitchell J. G, Subcontract: "A Flexible Base for Distributed Programming", SMLI technical report 93–13

[15] Hamilton G., Kougiouris P., "The Spring Nucleus: A Microkernel for Objects", *USENIX Summer Conference*, July 1993

[16] Khalidi, Y. A. and Nelson, M. N., "An Implementation of UNIX on an Object–oriented Operating System", *USENIX Winter Conference*, January 1993.

[17] Radia S., Powell M. P., "Generic Services and Type Parameterization", internal Spring document

[18] OMG, CORBA 1.1

[19] Radia, S. *Names, Contexts, and Closure Mechanisms in Distributed Computing Environments*, Ph.D. thesis, Univ. of Waterloo, Dept. of Comp. Sci., Waterloo, Ontario, Canada, 1989. Also report UW/ICR 90–01.

[20] Jul E., Levy H., Hutchinson N., Black A., "Fine–Grained Mobility in the Emerald System," 11th ACM Symposium on Operating Systems Principals. November 1987

[21] Hutchinson N., "Personal Communication,", 1993